A Scalable and Efficient Storage Allocator on Shared-Memory Multiprocessors

Voon-Yee Vee Wen-Jing Hsu

Centre for Advanced Information Systems, SAS
Nanyang Technological University
Nanyang Avenue, Singapore 639798

Abstract

An efficient dynamic storage allocator is important for time-critical parallel programs. In this paper, we present a fast and simple parallel allocator for fixed size block on shared-memory multiprocessors. We show both theoretically and empirically that the allocator incurs very low lock contention. The allocator is tested with parallel simulation applications with frequent allocation and release requests. The results confirm that our allocator is highly efficient and is scalable to more processors.

Keywords

Parallel storage allocators; minimization of lock contention; scalable parallel algorithms.

1 Introduction

We present an efficient special-purpose storage allocator for shared-memory multiprocessors. In particular, we emphasize on the time efficiency and the scalability of a parallel allocator. To justify the emphasis on time rather than space, it is noticed that the primary reason for most programmers to switch from uniprocessor to multiprocessor is to gain some speedups for the applications, rather than to look for larger memory capacity.

We have made the following assumptions:

1. *Shared-memory architecture*, which is based on a centralized memory that can be accessed by each processor directly.

2. *Frequent allocation and deallocation of blocks of fixed sizes.* For example, in parallel discrete event simulation (PDES) [5], a message or an event is often implemented as a fixed-size block. A PDES typically comprises a series of creation and processing of events, which require the allocation and release of fixed-size blocks respectively.

We introduce the algorithm of our parallel allocator in Section 2. Some analysis of the algorithm are also presented. We propose a very efficient implementation of the algorithm in Section 3. Section 4 compares the performance and the scalability of our allocator with some other parallel allocators.

2 The Algorithm

2.1 Some Considerations

Consider the execution of a parallel program in which allocation and release requests of fixed-size blocks are issued frequently. A global free list (or global pool) $\mathcal{G}$ of such blocks can be maintained in the memory. Exclusive access to $\mathcal{G}$ must be ensured to avoid data races. This can be realized by using a lock. This simple algorithm however is inadequate because allocation requests and release requests can never be serviced in parallel. Moreover, each such request always involves two lock operations—to acquire and to release a lock—which can be quite expensive. The performance can deteriorate further when there is a large number of processors, or the allocation and release requests are issued more frequently by the program.

The lock contention can be alleviated by maintaining a local free list (or local pool) lp_i to be accessed by processor p_i. In this arrangement, no lock operation needs to be involved for a process running on p_i to access lp_i. Lock operations will only be needed when a process accesses $\mathcal{G}$.

It is this concept of multiple pools that our algorithm is based on. Our algorithm, which will be further elaborated in the next section, has been carefully designed in order to achieve a very low lock contention even in the worst case condition. Furthermore, each allocation and release request can be serviced in almost constant time with a very small constant factor.

2.2 The Mechanism

There are two major components in our algorithm:
(1) A global pool $\mathcal{G}$. Exclusive access to the global pool is realized by using a lock.
(2) A set of P local pools, P being the total number of processors. Specifically, each processor p_i is assigned a local pool lp_i, which is initially empty. A process running on p_i is allowed to access only lp_i and $\mathcal{G}$. This guarantees exclusive access to each local pool.

The following parameters, whose values are positive integers, define the capacity of each local pool and the transfers between the global pool and local pools:

Definition 2.1.
- *m is the capacity of each local pool (i.e., it can hold at most m free blocks), where $m \geq 1$.*
- *m_α is the number of blocks per transfer from the global pool to any local pool, where $1 \leq m_\alpha \leq m$.*
- *m_β is the number of blocks per transfer from any local pool to the global pool, where $1 \leq m_\beta \leq m$.*

In the following description, we assume that all allocation and release operations are performed on processor p_i. The allocation procedure begins by checking if lp_i is empty. If lp_i is empty, the procedure requests m_α blocks from $\mathcal{G}$. It then returns a block from lp_i.

The release procedure begins by checking if lp_i is full, i.e., it holds m blocks. If lp_i is full, the procedure returns m_β blocks to $\mathcal{G}$. The block to be released is then returned to lp_i.

With this algorithm, it is easy to see that lock operations need not always be involved when servicing allocation or release requests. Moreover, the information related to a local pool lp_i can be cached by the processor p_i. This implies that the time taken to service a request is normally not affected by the services to other requests that are running in parallel on other processors.

In the next section, we examine the effects of the parameters m, m_α and m_β on the frequency of the lock operations. In particular, with the aim of minimizing the lock contention in the worst case condition, we obtain the relation $m_\alpha = m_\beta = \frac{m}{2}$ when m is even. We will introduce an efficient implementation of our algorithm which conforms to this relation in Section 3.

2.3 Minimizing the Lock Contention

Observe that the parameters m_α, m_β and m can affect the number of lock operations involved. For example, assume that a process running on processor p_i is going to issue n consecutive allocation requests. If lp_i is already empty and $m_\alpha = 1$, it can be verified that each of these n requests will involve lock operations (since lp_i will be empty after servicing every request). It can also be verified that under the same assumptions but $m_\alpha = k$, only $\lceil \frac{n}{k} \rceil$ of the requests will involve lock operations.

2.3.1 The Relation among m, m_α and m_β

We need to introduce a few notations before continuing our analysis. Consider only the services of allocation and release requests running on processor p_i. Let $\tau_{i,j}$ be the jth service which involves lock operations. Let $\phi_{i,j}$ be the number of services of allocation and release requests between $\tau_{i,j}$ and $\tau_{i,j+1}$ (exclusive). By definition, none of these services involves lock operations. Let ϕ_{min} denote the minimum value of $\phi_{i,j}$ for any i and j.

We first derive an upper bound of the total number of accesses to the global pool by all processors.

Lemma 2.2. *If processor p_i receives a total of S_i requests in executing a parallel program, it will make no more than $\lceil \frac{S_i}{\phi_{min}+1} \rceil$ accesses to the global pool.*

Proof. We divide the S_i requests into several partitions, where each consists of $\phi_{min} + 1$ requests, except the last partition which may consist of less than $\phi_{min} + 1$ requests. From the definition of ϕ_{min}, it is clear that in each partition there exists no more than one request that involves lock operations.

There is a total of $\lceil \frac{S_i}{\phi_{min}+1} \rceil$ partitions. This means that p_i will make no more than $\lceil \frac{S_i}{\phi_{min}+1} \rceil$ accesses to the global pool. ■

Theorem 2.3. *Assume that a parallel program makes a total of S requests. The total number of accesses to the global pool by all processors is bounded above by $\lfloor \frac{S}{\phi_{min}+1} \rfloor + P$.*

Proof. Assume that there are P processors $p_1, p_2, \ldots, p_P$, each receives $S_1, S_2, \ldots, S_P$ requests respectively. Let M_i be the total number of accesses to the global pool made by processor p_i, and $M = \sum_{i=1}^{P} M_i$ be the total number of accesses to the global pool made by all processors.

From Lemma 2.2, we have $M = \sum_{i=1}^{P} M_i = \sum_{i=1}^{P} \lceil \frac{S_i}{\phi_{min}+1} \rceil \leq \sum_{i=1}^{P} (\lfloor \frac{S_i}{\phi_{min}+1} \rfloor) + P \leq \lfloor (\sum_{i=1}^{P} S_i)/(\phi_{min} + 1) \rfloor + P = \lfloor \frac{S}{\phi_{min}+1} \rfloor + P$ ■

To minimize the lock contention, we need to minimize $M = \lfloor \frac{S}{\phi_{min}+1} \rfloor + P$. Since both S and P are fixed for a single run of a parallel program, the only choice is to maximize ϕ_{min}. Lemma 2.4 derives the relation among ϕ_{min}, m, m_α and m_β.

Lemma 2.4. $\phi_{min} = \min(m_\alpha, m_\beta, m-m_\alpha, m-m_\beta) - 1$. *In other words, if a processor has just accessed the global pool upon a request, it guarantees not to access the global pool when handling the next* $\min(m_\alpha, m_\beta, m-m_\alpha, m-m_\beta) - 1$ *requests.*

Proof. Consider two services $\tau_{i,j}$ and $\tau_{i,j+1}$, which both involve lock operations as defined previously. In the algorithm, a service $\tau_{i,j}$ will involve lock operations only under one of the following conditions: (a) $\tau_{i,j}$ services an allocation request when lp_i is empty, or (b) $\tau_{i,j}$ services a release request when lp_i is full. Therefore, we need to consider the following four cases:

Case 1. Both $\tau_{i,j}$ and $\tau_{i,j+1}$ belong to condition (a)

Case 2. $\tau_{i,j}$ and $\tau_{i,j+1}$ belong to conditions (a) and (b) respectively

Case 3. $\tau_{i,j}$ and $\tau_{i,j+1}$ belong to conditions (b) and (a) respectively

Case 4. Both $\tau_{i,j}$ and $\tau_{i,j+1}$ belong to condition (b)

Consider Case 1. The local pool lp_i has $m_\alpha - 1$ blocks just after $\tau_{i,j}$ and is empty just before $\tau_{i,j+1}$. In this case, the minimum of $\phi_{i,j}$ is achieved if p_i services consecutively m_α allocation requests just after $\tau_{i,j}$. Servicing the first $m_\alpha - 1$ requests reduces the number of blocks in lp_i from $m_\alpha - 1$ to 0 and does not involve any lock operation. $\tau_{i,j+1}$ will service the m-th request. Therefore, $\phi_{i,j} = m_\alpha - 1$.

Other cases can be analyzed similarly. Table 1 summarizes the results obtained.

Case	Lower bound of $\phi_{i,j}$
1	$m_\alpha - 1$
2	$(m - m_\alpha) - 1$
3	$(m - m_\beta) - 1$
4	$m_\beta - 1$

Table 1: Lower bounds of $\phi_{i,j}$ for various cases. For the proof of Lemma 2.4.

Therefore, by definition, ϕ_{min} is simply the minimum of the lower bounds of $\phi_{i,j}$ for the four various cases:

$$\begin{aligned} \phi_{min} &= \min(m_\alpha - 1, (m - m_\alpha) - 1, \\ &\qquad (m - m_\beta) - 1, m_\beta - 1) \\ &= \min(m_\alpha, m_\beta, m - m_\alpha, m - m_\beta) - 1 \end{aligned}$$

■

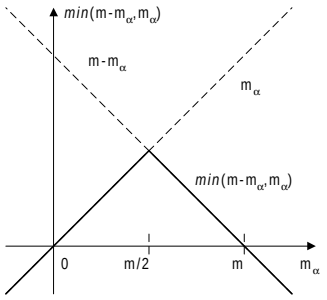


Figure 1: The value of $\min(m_\alpha, m - m_\alpha)$, which reaches its maximum when $m_\alpha = \frac{m}{2}$.

Theorem 2.5. *The algorithm incurs the lowest lock contention in the worst case when*

$$m_\alpha = m_\beta = \frac{m}{2} \quad \textit{when } m \textit{ is even}$$

and

$$\begin{cases} m_\alpha = \lceil \frac{m}{2} \rceil \text{ or } \lfloor \frac{m}{2} \rfloor \\ m_\beta = \lceil \frac{m}{2} \rceil \text{ or } \lfloor \frac{m}{2} \rfloor \end{cases} \quad \textit{when } m \textit{ is odd}$$

Proof. From Theorem 2.3 and Lemma 2.4, we need to maximize $\phi_{min} = \min(m_\alpha, m_\beta, m - m_\alpha, m - m_\beta) - 1$ in order to minimize the lock contention. It is clear that maximizing ϕ_{min} is equivalent to maximizing $\min(m_\alpha, m - m_\alpha)$ and $\min(m_\beta, m - m_\beta)$.

First consider the case when m is even. The value of $\min(m_\alpha, m - m_\alpha)$ is plotted in Figure 1. The peak value of the expression is obtained when $m_\alpha = \frac{m}{2}$. Similarly, $\min(m_\beta, m - m_\beta)$ reaches its maximum when $m_\beta = \frac{m}{2}$. Using a similar method to tackle the case when m is odd, the value of ϕ_{min} is found to reach its maximum when $m_\alpha = \lceil \frac{m}{2} \rceil$ or $\lfloor \frac{m}{2} \rfloor$, and $m_\beta = \lceil \frac{m}{2} \rceil$ or $\lfloor \frac{m}{2} \rfloor$. ■

2.3.2 Guidelines for Choosing m

We have considered the proper values of m_α and m_β for any given value of m in the previous section. To choose the proper value for m, we need to take the following two conflicting requirements into consideration:

1. From Theorem 2.5, we should maximize m in order to lower the lock contention.

2. It can be shown that our allocator requires at least $M + P \cdot m$ blocks in total for proper program execution, where M is the maximum number of blocks required by the program at any instant during the execution[1]. Clearly, the additional $P \cdot m$ blocks

[1] Assume that the program holds M blocks at time t. In the worst case, each local pool is full (i.e., holding m blocks) in time t. Summing these up, $M + P \cdot m$ blocks are required by the program.

represent an overhead, and we therefore should keep m as small as possible to reduce memory consumption.

The two goals above are conflicting with each other. We require small m to achieve good space efficiency and large m to achieve good time efficiency. It is however not difficult to choose a suitable m for typical applications given the underlying machine architectures. For example, if we have 16 processors, and each block has a size of 16 bytes. We can choose, say, $m = 1024$. This imposes additional $P \cdot m = 16 * 1024 = 16$k blocks, which is equivalent to $16\text{k} * 16\text{bytes} = 256$kbytes of memory. A 16-processor machine nowadays typically has at least a few gigabytes of memory. A payoff of only 256kbytes for a better performance on such machines is definitely worthwhile.

In the next section, we describe an efficient implementation which conforms to $m_\alpha = m_\beta = \frac{m}{2}$ when m is even.

3 An Efficient Implementation

3.1 Local Pools

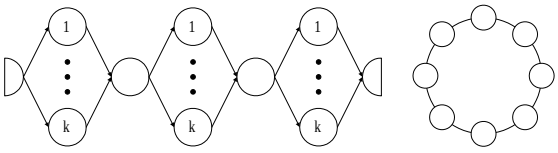
In this implementation, each local pool maintains the following components:

(1) Two stacks of free blocks. Each of the stacks is implemented as a linked list of free blocks. For convenience, they are referred to as *Active* stack and *Backup* stack respectively. Each stack is allowed to hold at most $\frac{m}{2}$ blocks. To support this implementation, each block is tagged with a pointer to another block.
We control the storage of the free blocks according to this principle: the Active stack is allowed to keep blocks if and only if the Backup stack is full. In other words, if the local pool has no more than $\frac{m}{2}$ blocks, all blocks will be kept in the Backup stack; if it has $k > \frac{m}{2}$ blocks, we keep $k - \frac{m}{2}$ blocks in the Active stack and $\frac{m}{2}$ blocks in the Backup stack.

(2) A counter recording the total number of memory blocks residing in the local pool. The counter helps the allocation and release procedures to decide quickly either the Active stack or the Backup stack it needs to access to.

3.2 Global Pool

The global pool maintains an array of pointers to several stacks of memory blocks. Each pointer refers to either a list of $\frac{m}{2}$ blocks or null.



(a) (b)

Figure 2: (a) A tandem queueing network and (b) a super-ping simulation.

The array is treated as a stack which allows push and pop operations. When a local pool requests for $\frac{m}{2}$ blocks from the global pool, it pops the top entry in the stack—which is a list of $\frac{m}{2}$ free blocks—to the local pool. When a local pool returns $\frac{m}{2}$ blocks to the global pool, it pushes the list of blocks onto this stack. Each push or pop operation takes a constant amount of time to complete. To avoid data races, lock mechanism is needed to coordinate the push and pop operations.

4 Performance

To evaluate the time efficiency, two parallel simulation programs which can be configured to make use of various allocators have been implemented. In a discrete event simulation, various events are generated and processed frequently. Therefore, the generation and removal of events must be done efficiently to reduce the time required to perform the simulation. This amounts to requiring an efficient implementation of allocation and release routines for fixed size blocks. Parallel simulation programs are thus realistic test suites for parallel allocators.

Both simulation programs are implemented using the Cilk language [1, 4], and adopt the safetime approach for parallel simulation proposed in [2].

1. Tandem Queueing Networks, as illustrated in Figure 2(a). We simulate a tandem network with 17 stages while each stage has 4 servers (i.e., $k = 4$) (which is a configuration used in [3] to benchmark a simulation algorithm).

2. Super-Ping Simulation, as illustrated in Figure 2(b). A total of 1000 LPs are simulated in our experiment.

The allocators which are evaluated in the experiment are:

1. [Method VH] The allocator using our algorithm. A value of $m = 512$ has been chosen. The global pool is initialized to hold P lists of free blocks while each list holds $\frac{m}{2}$ free blocks, where P is the number of processors.

The initial number of block lists in the global pool is set to be $P \cdot m$, P being the number of processors. Whenever the global pool runs out of free blocks, additional $\frac{P \cdot m}{2}$ blocks will be requested from the system[2].

2. [Method GP] Global free list (or global pool). Only one global free list is maintained. Exclusive access to the pool is realized by using a lock.

3. [Method CM] The Cilk allocator that is packaged along with the Cilk distribution [6]. The allocator adopts a variation of the *segregated free list* scheme [9]. In this scheme, an allocator maintains an array of free lists, each holding free blocks of a particular size. The current implementation maintains an array of free lists, and a lock is used to ensure atomic access to the lists. Therefore, lock operations will always be involved in servicing each allocation or release request.

4. Vo's `vmalloc()` allocator [8, 9]. The allocator is shown to be consistently among the fastest for a number of test applications. The allocator allows different "regions"—subsets of the overall heap memory—to be managed by different methods. In our empirical study, we define regions that follow the discipline to obtain raw memory with UNIX system call `sbrk()`, and use the special purpose method `Vmpool` which is suitable for allocating fixed size blocks efficiently. We have also made necessary modification to the source codes to parallelize the allocator.

   In the current implementation of `vmalloc()`, no region can be concurrently accessed. In order not to violate this constraint, we open for each processor p_i a region R_i. Consider the allocation and release requests received by p_i. When servicing an allocation request, p_i simply returns a block from R_i. For a release request, we have two choices: we can return the block either to the region where it is allocated from, or simply to R_i. They are elaborated below:

   (a) [Method VMSR] Return the block to the region where it is allocated from. To realize this approach, we maintain a free list F_i for every processor p_i, and a lock L_i to guard every access to each free list F_i in order to avoid race condition.

   When servicing a request of releasing block B, we first determine the region R_j where B is allocated from. If $i \neq j$, we push B to the free list F_j. Otherwise (i.e., $i = j$), we push B as well as all the blocks in F_i back to R_i. This guarantees that a block will always be released to the region where it is allocated from.

   (b) [Method VMDR] Return the block to the region associated with the processor. We maintain neither a free list nor a lock in this case, but simply return the block to be released to R_i. Since no lock operations and fewer instructions are involved as compared with the previous algorithm, this algorithm should be more time-efficient. However, since the algorithm does not guarantee that a block will always be released to the region where it is allocated from, we cannot use `vmcompact()` to release a region's free space back to the system.

[2]Since the programs are written in Cilk, in the current implementation we request larger space from the Cilk allocator.

4.1 Results

The simulation programs run on a 4 processor 250MHz UltraSPARC Enterprise 3000 Sun SMP with 512MB of memory. The results are summarized in Figure 3. The charts illustrate the average allocation time and the average deallocation time using various parallel allocators for the two simulation programs.

From the charts, it is clear VH is superior to other methods in almost all cases. It results in the fastest overall performance, followed by VMDR. In the experiments, VH is also among the most scalable allocators. With some of the methods (such as VH and VMDR), the time for servicing a request sometimes drops when the number of processors increases. This can probably be explained by the fact that some allocators spend extra time to manage a global pool (or global region) when running on one processor, while they spend less time to manage several local pools (or local regions) with less blocks per pool. Furthermore, it is likely that there are many more cache misses using a single processor than using multiple processors.

The experiments verifies the claim that our allocator is highly efficient and is scalable to more processors.

5 Conclusion

We have presented a fast, simple, and highly scalable parallel allocator for fixed size blocks. We have also analyzed the algorithm and showed that the lock contention incurred by the allocator is very low. Since the algorithm is notable for being fast and highly scalable, it is well suited for many time-critical applications. For

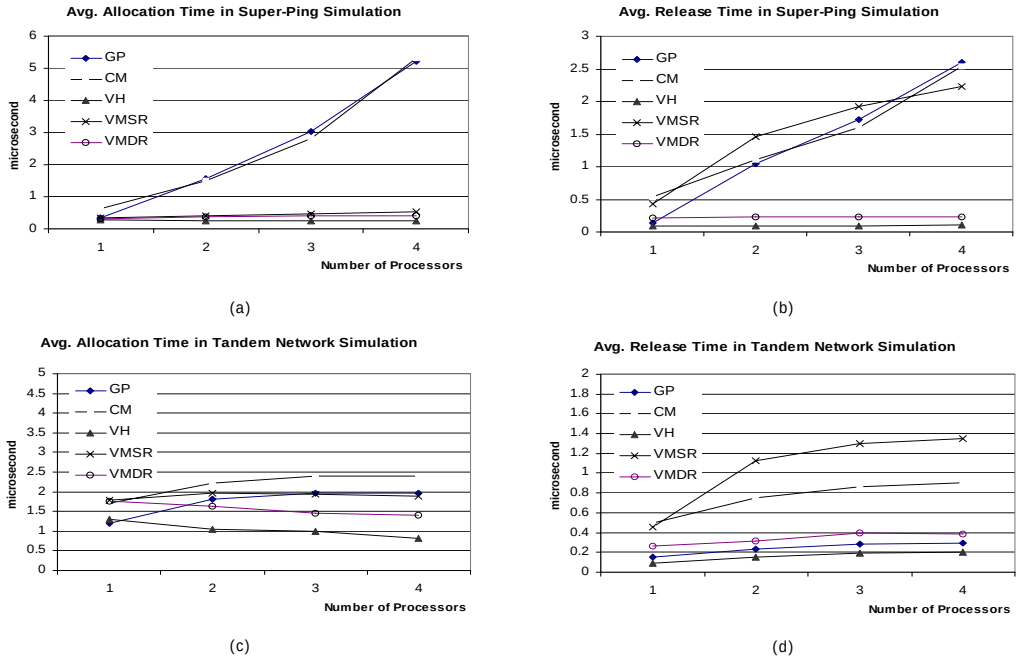


Figure 3: The average allocation time and release time for super-ping and tandem network simulations. The figures are measured using the Sun's high-resolution clock. Each average is computed from more than 100,000 samples.

example, the algorithm can be used in parallel simulation programs for event allocation and deallocation.

Our algorithm has been tested with two benchmark simulation applications. In the experiment, our allocator is compared with some of the existing allocators. The results show that our allocator has the least service time in most cases.

Acknowledgment

We wish to thank Charles Leiserson, for his generous provision of the Cilk environment and the fun of doing research using the great system. We also gratefully acknowledge the (anonymous) reviewers for their comments and the suggestions.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Honolulu, Hawaii, July 1995.

[2] W. Cai, E. Letertre, and S. J. Turner. Dag consistent parallel simulation: a predictable and robust conservative algorithm. In *Proceedings of 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, pages 178–181, Lockenhaus, Austria, June 10–13, 1997.

[3] R. Calinescu. Bulk synchronous parallel algorithms for optimistic discrete event simulation. Technical Report PRG-TR-8-96, Oxford University Computing Laboratory, 1996.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, June 17–19, 1998.

[5] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, Oct. 1990.

[6] MIT Laboratory for Computer Science. *Cilk-5.2 Reference Manual*, July 21, 1998. Available on the Internet from `http://theory.lcs.mit.edu/~cilk`.

[7] V.-Y. Vee and W.-J. Hsu. A scalable and efficient storage allocator on shared-memory multiprocessors. Technical Report CAIS-TR-98-22, Centre for Advanced Information Systems, Nanyang Technological University, Singapore, Dec. 1998. Available on the Internet from `http://www.cais.ntu.edu.sg:8000/tr.html`.

[8] K.-P. Vo. Vmalloc: A general and efficient memory allocator. *Software—Practice and Experience*, 26(3):357–374, Mar. 1996.

[9] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In H. G. Baker, editor, *Proceedings of International Workshop on Memory Management (IWMM'95)*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116, Kinross, Scotland, Sept. 1995.